Flot de compilation: propriétés pour l'analyse ou la transformation de code

Son Tuan Vu

Advisors: Karine Heydemann, Arnaud de Grandmaison, Albert Cohen

Team Alsoc
Laboratoire d'Informatique de Paris 6

January 31 2019

Outline

1 Introduction

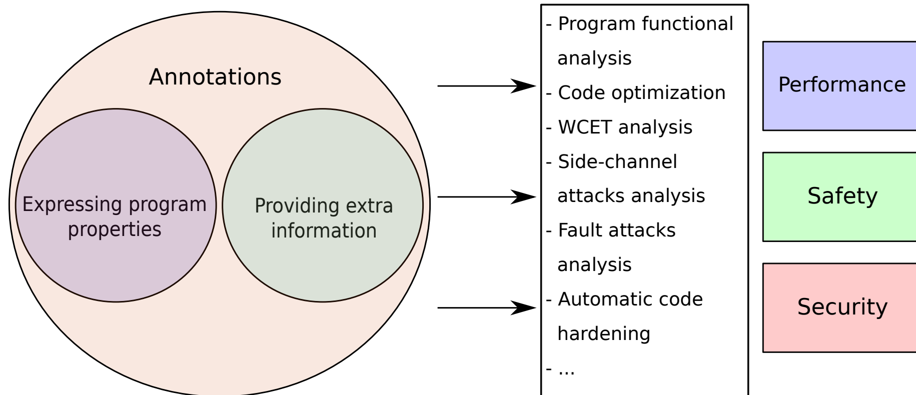
2 Proposed solutions

3 Conclusion

4 Bibliographie

Background and motivation

- *Annotations = program properties + extra information*
- Used in multiple application fields.



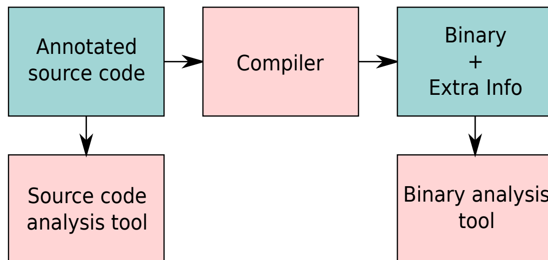
- Annotations are consumed by analysis tools or program transformation tools working from the source level to the binary level.

Related work

- Annotation languages :
  - GNU attributes, Microsoft's SAL, JML for Java, ACSL for C, ...
  - At source-level
- Usages :
  - augment compiler optimizations [NZ13]
  - automatic code hardening at compilation time [Hil14]
  - flow information for WCET analysis at binary level [LPR14] [SCG+18]

⇒ No annotation languages for all these properties

⇒ No compilers propagating annotations until the binary other than CompCert



Examples of property

Authentication code used in security against fault attacks field (from FISSC [DPP+16]).

```
1  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
2    int i;
3    int diff;
4    if (g_ptc > 0) {
5      diff = 0;
6
7      for (i = 0; i < 4; i++)
8        if (g_userPin[i] != g_cardPin[i])
9          diff = 1;
10
11     if (diff == 0) {
12       *g_ptc = 3;
13       return 1;
14     } else {
15       (*g_ptc)--;
16       return 0;
17     }
18   }
19   return 0;
20 }
```

Examples of property

Authentication code used in security against fault attacks field (from FISSC [DPP+16]).

```
1  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
2    int i;
3    int diff;
4    if (g_ptc > 0) {
5      diff = 0;
6
7      for (i = 0; i < 4; i++)
8        if (g_userPin[i] != g_cardPin[i])
9          diff = 1;
10
11     if (diff == 0) {
12       *g_ptc = 3;
13       return 1;
14     } else {
15       (*g_ptc)--;
16       return 0;
17     }
18   }
19   return 0;
20 }
```

- `g_cardPin` and `g_userPin` must not be modified.

Examples of property

Authentication code used in security against fault attacks field (from FISSC [DPP+16]).

```
1  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
2    int i;
3    int diff;
4    if (g_ptc > 0) {
5      diff = 0;
6
7      for (i = 0; i < 4; i++)
8        if (g_userPin[i] != g_cardPin[i])
9          diff = 1;
10
11     if (diff == 0) {
12       *g_ptc = 3;
13       return 1;
14     } else {
15       (*g_ptc)--;
16       return 0;
17     }
18   }
19   return 0;
20 }
```

- `g_cardPin` and `g_userPin` must not be modified.
- `verifyPIN` returns 1 only when `g_cardPin` and `g_userPin` match.

Examples of property

Authentication code used in security against fault attacks field (from FISSC [DPP+16]).

```
1  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
2    int i;
3    int diff;
4    if (g_ptc > 0) {
5      diff = 0;
6
7      for (i = 0; i < 4; i++)
8        if (g_userPin[i] != g_cardPin[i])
9          diff = 1;
10
11     if (diff == 0) {
12       *g_ptc = 3;
13       return 1;
14     } else {
15       (*g_ptc)--;
16       return 0;
17     }
18   }
19   return 0;
20 }
```

- g_cardPin and g_userPin must not be modified.
- verifyPIN returns 1 only when g_cardPin and g_userPin match.
- g_cardPin must be kept secret.

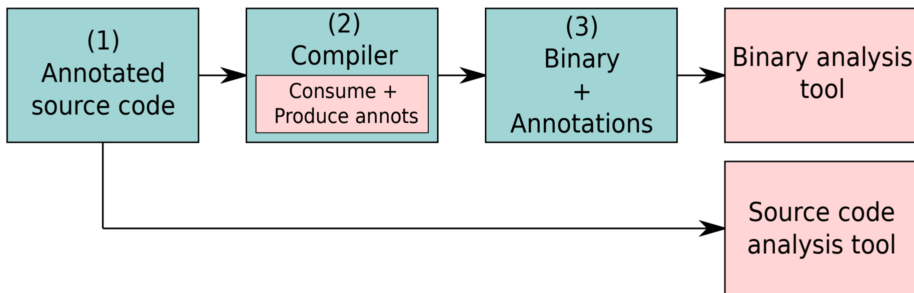
Examples of property

Authentication code used in security against fault attacks field (from FISSC [DPP+16]).

```
1  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
2    int i;
3    int diff;
4    if (g_ptc > 0) {
5      diff = 0;
6
7      for (i = 0; i < 4; i++)
8        if (g_userPin[i] != g_cardPin[i])
9          diff = 1;
10
11     if (diff == 0) {
12       *g_ptc = 3;
13       return 1;
14     } else {
15       (*g_ptc)--;
16       return 0;
17     }
18   }
19   return 0;
20 }
```

- g_cardPin and g_userPin must not be modified.
- verifyPIN returns 1 only when g_cardPin and g_userPin match.
- g_cardPin must be kept secret.
- For loop (line 7-9) must be executed exactly 4 times.

Problem statement



1. Design a source-level annotation language to express a wide range of properties.
2. Build a compilation framework which propagates annotations through an optimizing pipeline.
3. Define binary-level representation for the source-level annotation language.

Outline

1 Introduction

2 Proposed solutions
- Source-level annotation language
- Binary-level representation of the annotation language
- Annotation propagation in a compilation flow

3 Conclusion

4 Bibliographie

Outline

1 Introduction

2 Proposed solutions
- Source-level annotation language
- Binary-level representation of the annotation language
- Annotation propagation in a compilation flow

3 Conclusion

4 Bibliographie

Annotation language by example

- ACSL already allows specifying program functional properties.

```
1  #define ANNOT(s) __attribute__((annotate(s)))
2
3  // Function annotation
4  ANNOT("\\assigns g_ptc;"
5        "\\ensures \\result == 1 &&"
6                 " \\forall i; 0 <= i < 4: g_userPin[i] == g_cardPin[i];"
7        "\\ensures \\result == 0 &&"
8                 " \\exists i; 0 <= i < 4: g_userPin[i] != g_cardPin[i];")
9  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
10   int i;
11   int diff;
12   if (g_ptc > 0) {
13     diff = 0;
14
15     for (i = 0; i < 4; i++)
16       if (g_userPin[i] != g_cardPin[i])
17         diff = 1;
18
19     if (diff == 0) {
20       *g_ptc = 3;
21       return 1;
22     } else {
23       (*g_ptc)--;
24       return 0;
25     }
26   }
27   return 0;
28 }
```

Annotation language by example

- Introduce semantic keywords (e.g. \secret) to specify non-functional properties, used by analysis tools.

```
1  #define ANNOT(s) __attribute__((annotate(s)))
2
3  // Variable annotation
4  int verifyPIN(ANNOT("\\invariant \\secret()") char *g_cardPin,
5                char *g_userPin, int *g_ptc) {
6    int i;
7    int diff;
8    if (g_ptc > 0) {
9      diff = 0;
10
11     for (i = 0; i < 4; i++)
12       if (g_userPin[i] != g_cardPin[i])
13         diff = 1;
14
15     if (diff == 0) {
16       *g_ptc = 3;
17       return 1;
18     } else {
19       (*g_ptc)--;
20       return 0;
21     }
22   }
23   return 0;
24 }
```

Annotation language by example

- Introduce semantic variables (e.g. \count) to specify flow information for example.

```
1  #define ANNOT(s) __attribute__((annotate(s)))
2
3  int verifyPIN(char *g_cardPin, char *g_userPin, int *g_ptc) {
4    int i;
5    int diff;
6    if (g_ptc > 0) {
7      diff = 0;
8
9      // Statement annotation
10     prop1: ANNOT("\\ensures \\count() == 4;")
11     for (i = 0; i < 4; i++)
12       if (g_userPin[i] != g_cardPin[i])
13         diff = 1;
14
15     if (diff == 0) {
16       *g_ptc = 3;
17       return 1;
18     } else {
19       (*g_ptc)--;
20       return 0;
21     }
22   }
23   return 0;
24 }
```

Annotation language summary

- $Annotation = annotated\ entity \land predicate \land referenced\ variables$
- $Annotated\ entity = function \lor variable \lor statement$

Outline

1 Introduction

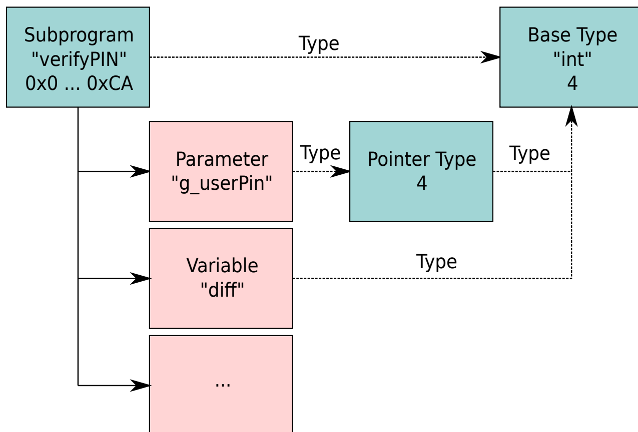
2 Proposed solutions
- Source-level annotation language
- Binary-level representation of the annotation language
- Annotation propagation in a compilation flow

3 Conclusion

4 Bibliographie

DWARF debugging format

- Widely used and extensible.
- Represent the executable program using a tree of descriptive entities called *Debugging Information Entries* (DIEs).



Extending DWARF debugging format

- Introduce new DIEs to represent annotations and semantic variables.
- Examples :

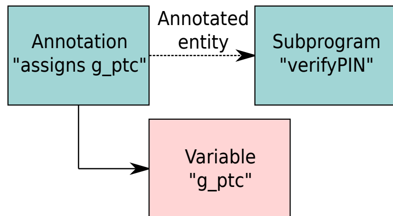


Figure: Function annotation

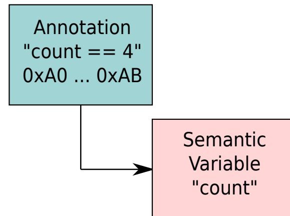


Figure: Statement annotation

Outline

1. Introduction
2. Proposed solutions
   - Source-level annotation language
   - Binary-level representation of the annotation language
   - Annotation propagation in a compilation flow
3. Conclusion
4. Bibliographie

LLVM Overview

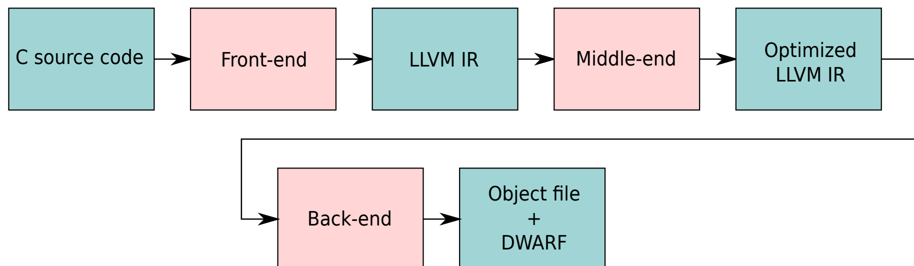
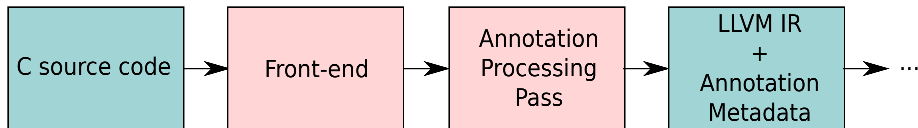


Figure: Framework overview

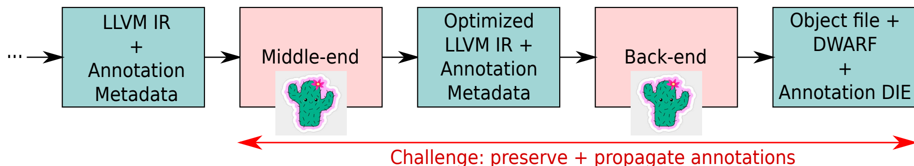
- LLVM already provides metadata mechanism to convey extra information about the code (e.g. debug information, loop unrolling factor, etc.).
- Only debug information metadata is preserved, propagated and emitted into the binary : other metadata is discarded at the beginning of the back-end.

Annotation representation in LLVM

- Annotation = dedicated LLVM generic metadata (new) :
  - Annotated entity
    - Variable + Function = debug info metadata (existing)
    - Statement = delimited by intrinsics begin-end (new)
  - Referenced variables
    - Variable = debug info metadata (existing)
    - Semantic variable = dedicated LLVM generic metadata (new)



Annotation propagation in LLVM

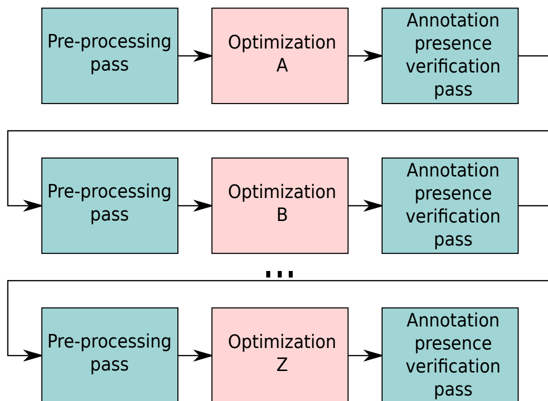


- Examples of annotation propagation problem :
  - Annotated IR instructions merged or reordered or removed.
  - Loop counters throughout loop optimizations [LPR14].
  - Variables annotated or referenced in an annotation get optimized out.
- Examples of debugging information propagation bug :
  - `mem2reg` pass : debug location of `PHINodes`.
  - `LiveDebugVariables` pass : location ranges of auto variables.

$\Rightarrow$ Build the minimal pipeline : start with no optimization (O0), progressively add (annotation-aware) optimizations to the pipeline (towards O1).

Verification of annotation presence throughout optimizations

To detect problems when propagating annotations :
⇒ mechanism to verify the *presence* of annotations throughout the optimizing pipeline.



Tests

- Benchmarks considered :
  - codePin : `verifyPIN` is annotated to make sure that it correctly grants access rights to an user under fault attacks.
  - First-order masked AES [HOM06] : `plain_text` and `round_key` are annotated as secret variables, masked variables are also annotated.
  - RSA (from FISSC) and SHA (from MiBench [GRE+01]) : simply annotate some random functions and data objects defined in the source program, without specifying real security properties.
- Function + variable annotations :
  - compiled at LLVM *-O1*
  - verifying manually the annotation correctness
  - annotations found in DWARF section but variable debug info may be erroneous
- Statement annotation : implementation in progress.

Outline

1 Introduction

2 Proposed solutions

3 Conclusion

4 Bibliographie

Conclusion

- Summary :
  - Defined an annotation language for C programs based on ACSL, capable of expressing a wide range of properties from several application fields.
  - Defined a representation in the binary of the source-level annotations by extending the DWARF format.
  - Proposed mechanisms to propagate and verify annotations throughout an optimizing compilation pipeline.
- Perspectives :
  - Propose an automatic *correctness* validation process
  - Measure the impact of the minimal pipeline and the annotation
  - Show the utility of the annotation propagation

Thank you for your attention !

Outline

1 Introduction

2 Proposed solutions

3 Conclusion

4 Bibliographie

Bibliographie I

Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens.
Fissc : A fault injection and simulation secure collection.
pages 3–11, 09 2016.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown.
Mibench : A free, commercially representative embedded benchmark suite.
In *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, WWC '01, pages 3–14, Washington, DC, USA, 2001. IEEE Computer Society.

Christoph Hillebold.
Compiler-assisted integrits against fault injection attacks.
Master's thesis, University of Technology, Graz, December 2014.

Christoph Herbst, Elisabeth Oswald, and Stefan Mangard.
An aes smart card implementation resistant to power analysis attacks.
In *Proceedings of the 4th International Conference on Applied Cryptography and Network Security*, ACNS'06, pages 239–252, Berlin, Heidelberg, 2006. Springer-Verlag.

Hanbing Li, Isabelle Puaut, and Erven Rohou.
Traceability of flow information : Reconciling compiler optimizations and wcet estimation.
In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 97 :97–97 :106, New York, NY, USA, 2014. ACM.

Bibliographie II

Kedar S. Namjoshi and Lenore D. Zuck.
Witnessing program transformations.
In Francesco Logozzo and Manuel Fähndrich, editors, *Static Analysis*, pages 304–323, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

Bernhard Schommer, Christoph Cullmann, Gernot Gebhard, Xavier Leroy, Michael Schmidt, and Simon Wegener.
Embedded Program Annotations for WCET Analysis.
In *WCET 2018 : 18th International Workshop on Worst-Case Execution Time Analysis*, volume 63, Barcelona, Spain, July 2018. Dagstuhl Publishing.